Spring Boot In 3 Weeks

Week 1: Fundamentals
Week 2: Persistence
Week 3: Spring MVC

Contact Info

Ken Kousen

Kousen IT, Inc.

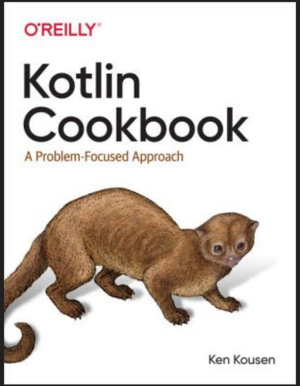
ken.kousen@kousenit.com

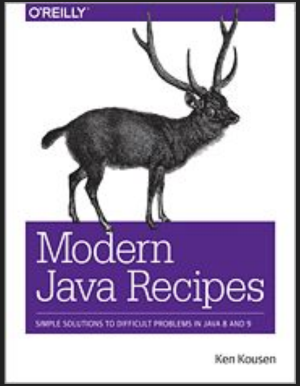
http://www.kousenit.com

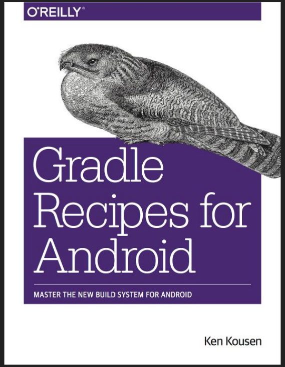
http://kousenit.org (blog)

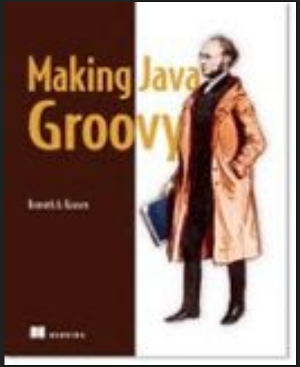
@kenkousen (twitter)

https://kenkousen.substack.com (newsletter)





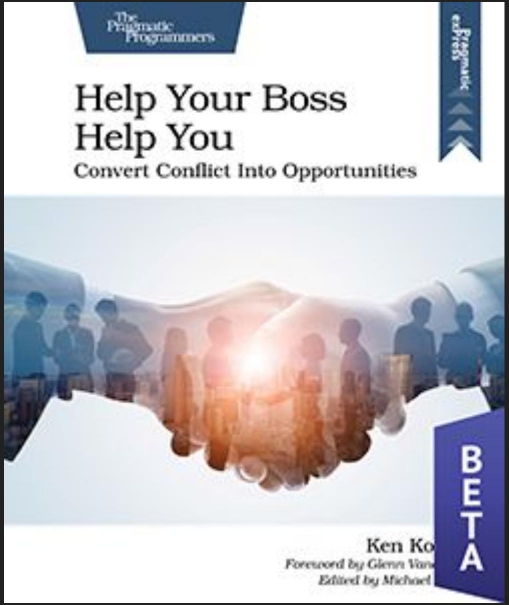

O'REILLY
Gradle Recipes for Android
MASTER THE NEW BUILD SYSTEM FOR ANDROID
Ken Kousen



New Book

Help Your Boss Help You

https://pragprog.com/titles/kkmanage/help-your-boss-help-you/



Week 1: Fundamentals

- Spring infrastructure
  - Dependency injection
  - Application Context
- Spring Boot
  - Starters
  - Auto-configuration
  - Component scan
- Rest clients and services
- Testing
  - Unit
  - Integration
  - Functional

Spring

Project infrastructure

Spring

- Lifecycle management of "beans"
    - Any POJO with getters/setters

Spring

- Provides "services"
  - transactions, security, persistence, …

Spring

- Library of beans available
  - transaction managers
  - rest clients
  - DB connection pools
  - testing mechanisms

Spring

- Need "metadata"
  - Tells Spring what to instantiate and configure
    - XML → old style
    - Annotations → used for standard components
    - JavaConfig → used for user-supplied beans
  - All still supported

Spring

- Application Context
  - Collection of managed beans
  - the "lightweight" Spring container

Spring Boot

- Easy creation and configuration for Spring apps
  - Many "starters"
  - Gradle or Maven based
- Automatic configuration based on classpath
  - If you add JDBC driver, it adds DataSource bean

Spring Initializr

Website for creating new Spring (Boot) apps

http://start.spring.io

Incorporated into major IDEs

Select features you want

Download zip containing build file

Spring Boot

Application with main method created automatically

Annotated with `@SpringBootApplication`

Gradle or Maven build produces executable jar in build/libs folder

```
    $ java -jar appname.jar
```

Or use gradle task `bootRun`

Spring MVC

Annotation based MVC framework

`@Controller` → controllers

`@GetMapping` → annotations for HTTP methods

`@RequestParam` and more for model parameters

Model interface → map for carrying data from one resource to another

Rest Client

Spring includes a class called `RestTemplate`

- Access RESTful web services
- Set HTTP methods, headers, query string, templates
- Use `RestTemplateBuilder` to create one
- Use content negotiation to return JSON or XML
- Convenient `getForObject(url, class)` method

Newer reactive client: WebClient

Logging

Spring libraries include SLF4J automatically

- Use `LoggerFactory.getLogger(... class name ...)`
- Returns an `org.slf4j.Logger` instance
- Invoke logging methods as usual

Dependency Injection

- Spring adds dependencies on request
  - Annotate field, or setter, or constructor
  - `@Autowired` → autowiring by type
  - `@Resource` (from Java EE) → autowiring by (bean) name, then by type if necessary

Testing

Spring tests automatically include special JUnit 5 extension

- `@ExtendWith(SpringExtension.class)`
- Annotate test class with `@SpringBootTest`
- Annotate tests with `@Test`
- Use normal asserts as usual

Unit Testing

Instantiate class and invoke methods

Dependencies can be mocked → Mockito is already included

Fast, but least realistic

Integration Testing

Special annotations for web integration tests

Uses Spring, but not an actual server

`@WebMvcTest(... controller class ...)`

`MockMvc` package

`MockMvcRequestBuilders`

`MockMvcRequestMatchers`

Functional Testing

Run on an actual test server

```
@SpringBootTest(webEnvironment = RANDOM)
```

- Spring chooses random port
- Deploys app
- Runs tests
- Shuts down server

Most realistic, but potentially slow

Parsing JSON

- Several options, but one is the `Jackson JSON 2` library
  - Create classes that map to JSON response
  - `restTemplate.getForObject(url, … your class …)`
- Maps JSON to Java objects

Component Scan

Spring detects annotated classes in the expected folders

`@Component` → Spring bean

`@Controller`, `@Service`, `@Repository` → based on `@Component`

Application properties

Two options for file name

Default folder is `src/main/resources`

- `application.properties` → standard Java properties file
- `application.yml` → YAML format

Summary for Week 1

Spring:

- Dependency injection
- Provides services
- Includes large API

Spring Boot:

- Used to create a new Spring app
- Auto-configures many beans

Great for web apps, restful web services, and more

Week 2: Persistence

JdbcTemplate → Pass SQL to DB

JPA → Use Java Persistence API

Spring Data JPA → Generate your entire DAO layer

Persistence

- Spring provides `JdbcTemplate`
    - Easy to access and use relational databases
    - Best if you already have the SQL you want to use

Persistence

More conventions:

Two standard files in `src/main/resources`

`schema.sql` → create test database

`data.sql` → populate test database

Both executed on startup, using DB connection pool

JdbcTemplate

- Standard practice:
  - Create DAO interface and implementation class
  - Autowire `DataSource` into constructor
  - Instantiate `JdbcTemplate` from `DataSource`
- Spring Boot lets you autowire the `JdbcTemplate` directly

JdbcTemplate

Use `queryForObject` to map DB row to Java class

(`query` method does the same for all rows)

In Java 7, uses inner class that implements `RowMapper<MyClass>`

In Java 8, can use lambda expression

H2 Database

- Add the H2 dependency
  - `runtime('com.h2database:h2')`
  - Automatically adds DataSource for it

If you add the web starter and the dev-tools dependency,

H2 console: http://localhost:8080/h2-console

DB URL in console of the form `jdbc:h2:mem:<generated>`

Or set `spring.datasource.generate-unique-name` to false

SimpleJdbcInsert

Specify table name and generated key columns

Create a `SqlParameterSource` or a `Map`

Run `executeAndReturnKey(parameters)`

Transactions

Spring transactions configured with `@Transactional`

Spring uses `TransactionManager` to talk to resource

- usually a relational DB, but other options available

@Transactional

Each method wrapped in a REQUIRED tx by default

Propagation levels:

- REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED

In tests, transactions in test methods roll back by default

Can configure isolation levels:

- READ_UNCOMMITTED, READ_COMMITTED,
- REPEATABLE_READ, SERIALIZABLE

JPA

- Java Persistence API
  - Uses a "provider" → Hibernate most common
- Annotate entity classes
  - `@Entity, @Table, @Column, @Id, @GeneratedValue`
- use in Spring `@Repository` → exception translation
  - `@PersistenceContext` → `EntityManager`

Spring Data

Large, powerful API

Create interface that extends a given one

  `CrudRepository, PagingAndSortingRepository`

  We'll use `JpaRepository<class, serializable>`

Add your own finder method declarations

All SQL generated automatically

Summary for Week 2

Persistence:

- JdbcTemplate, SimpleJdbcInsert
- `@PersistenceContext` for JPA
- Spring Data JPA → generate entire DAO layer

Transactions:

- `@Transactional` annotation
- Can set isolation level and propagation levels

Week 3: Spring MVC

- MVC libraries
- Error handling and bean validation
- Handler mappings, view resolvers, content negotiation
- CORS
- Profiles
- Annotated controllers vs functional endpoints

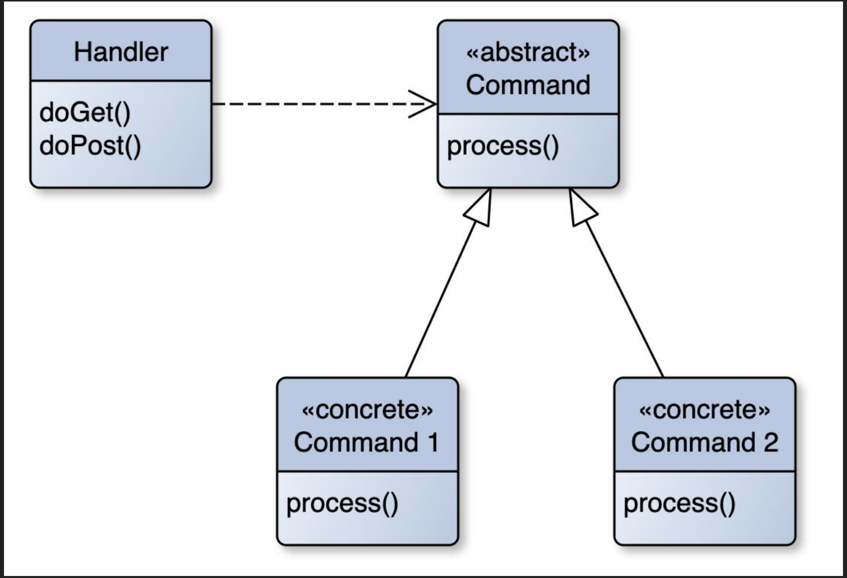
Spring MVC Starters

Add either web and/or webflux starter

- spring-boot-starter-web → Spring MVC
- spring-boot-starter-webflux → Reactive Spring
  - Need this for WebClient

Spring MVC

Designed around Front Controller design pattern



DispatcherServlet

- Central servlet that acts as front controller
  - Spring Boot sets up and maps automatically

Special Beans

Spring library useful bean types

- `HandlerMapping` and `HandlerAdapter` → maps URLs to bean methods
- `ViewResolver` → Converts strings to views
- `HandlerExceptionResolver` → Map exceptions/errors to views

Processing Requests

DispatcherServlet:

- Find WebApplicationContext and bind request
- Use locale resolver, if necessary
- Use theme resolver, if necessary
- Use multipart file resolver, if necessary
- Use HandlerMapping to invoke method
- Use ViewResolver to connect to view

Request Processing

Everything can be configured and customized

Spring Boot Simplifications

Spring Boot autoconfiguration provides:

- ContentNegotiatingViewResolver
- BeanNameViewResolver
- HttpMessageConverters
- Static content:
    - /static or /public or /resources directory

Path Matching and Content Negotiation

Disables suffix pattern matching by default

Uses Accept headers for content negotiation

Template Engines for dynamic HTML content

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

All use /src/main/resources/templates by default

Spring MVC

Annotation based MVC framework

`@Controller, @RestController` → controllers

`@GetMapping` → annotations for HTTP methods

Similar for POST, PUT, PATCH, DELETE, ...

`@RequestParam` and more for model parameters

`@PathVariable` for URI templates

Custom Error Page

In folder `src/main/resources/public/error`

- Add `404.html` (or other error code)
- More general, add 5xx.html, etc

CORS

Cross-origin resource sharing

Easy way: `@CrossOrigin` annotation

More complex: Register a `WebMvcConfigurer` bean
with `addCorsMappings(CorsRegistry)` method

Mock Objects

- Includes Mockito
- `@MockBean`
    - Set expectations and verify as usual

Application properties

Two options for file name

Default folder is `src/main/resources`

- `application.properties` → standard Java properties file
- `application.yml` → YAML format

Web Apps

- Add `Model` parameter to controller methods
  - Carries data from controllers to views
  - Model attributes copied into each request

Validation

- Spring uses any JSR-303 implementation on classpath
  - Hibernate validator by default
  - `@Valid`
  - `@Min`, `@Max`, `@NotBlank`, ...

Persistence

More conventions:

Two standard files in `src/main/resources`

`schema.sql` → create test database
`data.sql` → populate test database

Both executed on startup, using DB connection pool

application.properties:

spring.datasource.schema, spring.datasource.data
spring.sql.init.schema-locations, spring.sql.init.data-locations (Boot 2.5+)

Profiles

Create the same beans to be used under different situations

Either:

- Multiple files with profile name in them
  - application-{profilename}.properties

Or:

- One YAML file with section separated by ---

Profiles

```
logging:
	level:
		org.springframework.web: DEBUG
---
spring:
	profiles: prod
	datasource: …
---
spring:
	profiles: dev
	datasource: ...
```

Profiles

Annotate beans for specific profiles

```
@Profile("dev")
@Profile({"dev","prod"})
@Profile("!test")
```

Set the active profile:

- `spring.profiles.active = prod`
- Set `SPRING_PROFILES_ACTIVE` environment variable
- `--spring.profiles.active=prod` on command line

Web.fn

- Functional approach
  - Router function bean
    - maps URLs to handler methods
    - Kotlin has a nice DSL for it
  - Hander class
    - all methods take ServerRequest and return ServerResponse

Summary of Week 3

- Profiles
  - Different beans with same name under different profiles
  - Load beans conditionally
- Two different web libraries
  - web → MVC
  - webflux → Reactive
  - Can be used together
- Dispatcher servlet, handler mappings, view resolvers, content negotiation
- CORS
- Custom error handling
- Bean validation
- Annotated controllers vs functional endpoints